Recent Trends in Efficient Inference of Deep Learning Models on Edge Devices: A Survey

PANDRANGI ADITYA SRIRAM, Indian Institute of Technology, Hyderabad, India
RAMBHA SATVIK, Indian Institute of Technology, Hyderabad, India

Edge AI systems—from smartphones and IoT sensors to autonomous vehicles—must execute increasingly complex DNNs under severe memory, energy, and latency constraints. A growing body of work addresses this challenge along two complementary axes: system-level resource management (e.g., selective offloading), software optimizations (e.g. ViTALiTy), and memory-/compute-centric optimizations (e.g., swapping, pipelining, and specialized accelerators). In this joint presentation, we survey four representative frameworks spanning these trends:

- **Active-Reconfigurable-Array ViT Accelerator** (Shan & Zheng, 2023 [4]): a hardware accelerator tailored for Vision Transformers, leveraging a reconfigurable array to exploit ViT's attention patterns for high throughput and low power on-device. This yielded better accuracy, lower hardware (suitable for edge devices) and lower latency than baseline and other latest research.
- **Power Efficient ASIC Design for Vision Transformer using a symmetric matrix multiplier of matrices matrices** (Naveen. et al.[3], 2024): another hardware tailored for Vision Transformers (ViTs) which leverages a multi-head self-attention mechanism to obtain global image features, but this attention is computationally demanding. The authors achieve power-efficient ViT inference for edge deployment. This paper addresses this challenge by algorithmically simplifying the attention mechanism and designing a custom ASIC accelerator to implement it.
- **SwapNet** (Wang et al.[6], 2024) a middleware that enables large CNNs to run beyond an edge device's DRAM/VRAM budget by partitioning models into blocks and performing zero-copy, unified-memory swapping between SSD and GPU memory, with in-place pointer-based assembly to eliminate redundant copies.
- **Hermes** (Lee et al.[2], 2024) a runtime that pipelines transformer inference across CPU threads through parallel "Loading Agents," an "Inference Agent," and a "Daemon Agent" to load, execute, and free per-layer weights on the fly—achieving 60–90% peak memory reduction with up to 8× latency improvement for encoder-only models.

For each framework, we present (i) the core problem it addresses, (ii) its architectural or hardware innovations, and (iii) its evaluation results under representative edge-device settings. We conclude by comparing how these approaches trade off memory footprint, latency, accuracy, and energy, and we highlight open challenges in unifying system-level and hardware-level optimizations for scalable, low-power edge inference.

ACM Reference Format:
Pandrangi Aditya Sriram and Rambha Satvik. 2025. Recent Trends in Efficient Inference of Deep Learning Models on Edge Devices: A Survey. 1, 1, Article 1 (April 2025), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Pandrangi Aditya Sriram, Indian Institute of Technology, Hyderabad, India; Rambha Satvik, Indian Institute of Technology, Hyderabad, India.

ACM XXXX-XXXX/2025/4-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 POWER EFFICIENT ASIC DESIGN FOR VISION TRANSFORMER USING SYSTOLIC ARRAY MATRIX MULTIPLIER

1.1 Motivation

In edge environments, there is typically a tight power constraint. Vision Transformers are gaining huge attention in the field of Computer Vision due to their accuracy. ViTs employ MHA which is very expensive and power-hungry. A majority of the time is spent on softmax computation, as shown in the figure. The paper proposes a custom ASIC which seeks to optimize both power and latency while maintaining the accuracy.

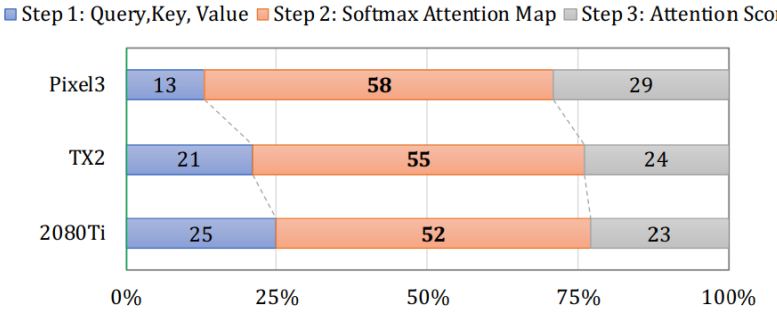

Fig. 1. Runtime Simulation of MHA on various devices [1]

1.2 Background on Vision Transformers and Terminology

The overall ViT encoder organization is illustrated in the following figure: patch embeddings (including a `[CLS]` token) are fed into a transformer encoder block, followed by a final MLP head for classification. Positional embeddings are added to the vectorized patches to tell the position of the patch to the model. The transformer itself consists of alternating attention layers with residual connections across them; alongside feedforward neural networks to process the relevant embeddings captured by the transformer.

1.3 Innovation

The main contributions of this paper are as follows:

- A **software optimization** replacing the costly softmax in attention with a Taylor expansion of the first order (as in ViTALiTy), reducing computational complexity
- A **hardware design** for self-attention using a Systolic Array Matrix Multiplier (SAMM) that accelerates the large matrix multiplications in multi-head attention.
- The use of **low-power techniques**, such as clock gating, in the ASIC implementation, yielding roughly 25.8% dynamic power reduction with modest area/frequency overhead.

1.3.1 Software and Mathematical Optimizations. A vision transformer takes an input image $x \in \mathbb{R}^{H\times W\times C}$ reshaped into patches $x \in \mathbb{R}^{N\times P^2\times C}$ where P is the number of patches along each dimension, C is the number of channels, H and W are the height and width, and N is the dimension of each of the P^2 patches, and $N = \frac{HW}{P^2}$. It has several layers of attention and feedforward neural networks, and it finally outputs an encoding of the patches that can be used for classification. Transformers utilize Query-Key-Value attention which are computed as follows for each head in

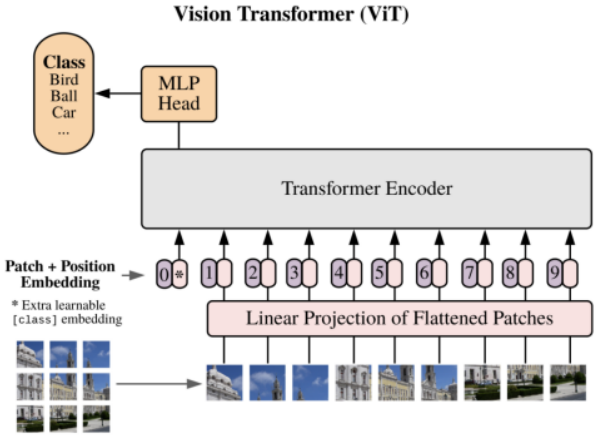

Fig. 2. Architecture of a Vision Transformer [3]

multi-headed attention:

$$Q = XW_Q \tag{1}$$

$$K = XW_K \tag{2}$$

$$V = XW_V \tag{3}$$

$$\text{Attention} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4}$$

This operation is heavily utilized in each of the many attention layers. Thus, a custom power-efficient ASIC was designed for this purpose. Softmax was observed to consume the maximum amount of time [1].

Three self-attention architectures are compared. The standard ViT uses softmax-based attention with quadratic complexity $O(n^2d)$ where d is the dimension of key vector. One such architecture, SimA eliminates softmax by L_1-normalizing the query and key, achieving linear complexity in sequence length. ViTALiTy replaces softmax with the first-order Taylor series attention and performs mean-centering of the keys (in order to reduce the size of softmax input), thus yielding linear complexity. Experimentally, ViTALiTy achieves similar accuracy (95.7% on MNIST) but with far fewer FLOPs and 37.7% faster inference than vanilla ViT. This efficiency motivates focussing the hardware design on the ViTALiTy model. The first-order Taylor series with row-mean centering as the low-rank component to linearize the cost of attention blocks is utilized (here, the hat represents row-mean centering). Mean-centering does not change the output of the softmax function and is taken advantage of.

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \approx \text{diag}^{-1}\left(n\sqrt{d}\mathbf{1}_n + Q\hat{K}_{sum}^T\right) \times \left(\sqrt{d}\mathbf{1}_n\mathbf{1}_n^T + Q\hat{K}^T\right) \tag{5}$$

1.4 Hardware Architecture

The ASIC implements the ViTALiTy attention using a **Systolic Array Matrix Multiplier (SAMM)**. The key components are:

- **Processing Units (PUs):** Each PU contains a multiplier and an adder (a MAC unit). PUs are tiled into a 2D grid as a pipelined systolic array. They stream data in a wave-like fashion for each matrix multiplication. Three such arrays handle the query-key, value, and output multipliers (sizes 17×32, 16×32, and 17×1). Systolic arrays provide high throughput and energy-efficient operation.
- **Column Accumulator:** Implements column-wise mean-centering of the key matrix before softmax. A dedicated column-accumulator unit performs this "mean centering of the inputs", obviating a separate softmax step.
- **Controller and Memory:** Control units control the data flow through various instructions. They are crucial for selection of weights, data reuse and reducing power consumption as well as memory accesses, while also ensuring seamless data transfer between memory and modules. On-chip memory buffers store the embedded input patches and the weight matrices (W_Q, W_K, W_V). The pipeline consists of a pre-processing stage which involves patch embedding, positional encoding, and weight quantization to 16-bit fixed point (to reduce power consumption with minimal drop of accuracy), the SAMM-based attention compute stage, and a post-processing MLP for classification.

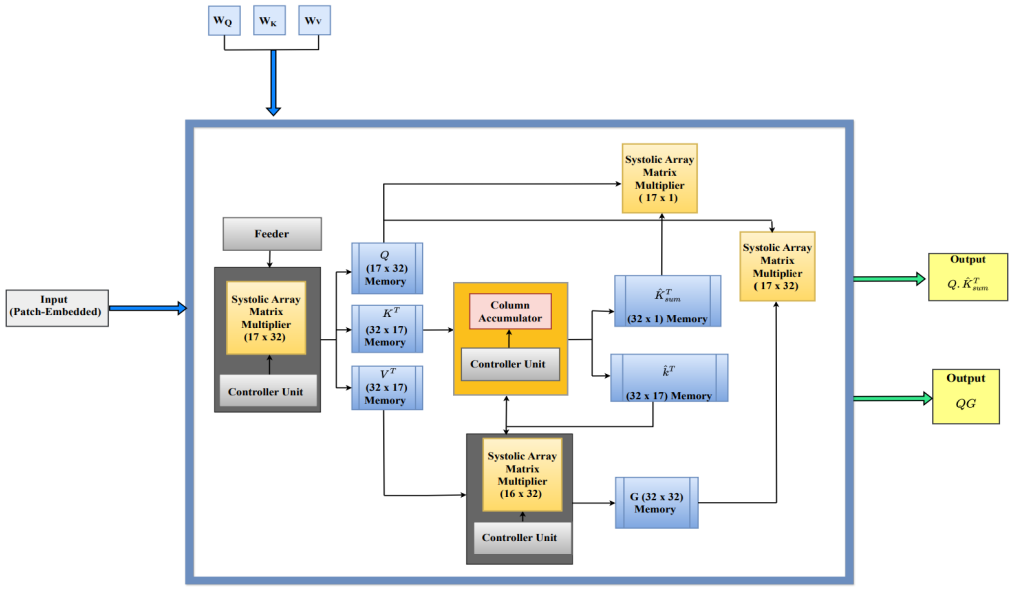

Fig. 3. Power Efficient ASIC Hardware Architecture [3]

1.5 Implementation and Results

The design was implemented in 16-bit fixed-point and synthesized using a 180 nm library at 100 MHz. Key results include:

- **Power Efficiency:** The SAMM accelerator consumes less power than prior designs. Applying clock gating (selective stopping of clock signals to inactive digital modules) cuts dynamic power by about 25.8% compared to baseline DianNao, and over 50% compared to Eyeriss.
- **Area and Speed:** At 180 nm the chip meets 100 MHz. A 90 nm simulation ran at 250 MHz with a smaller area, illustrating expected process-area-frequency trade-offs.

- **Accuracy:** Verification on MNIST showed the ASIC matches the baseline accuracy of 95% while using significantly less power and area.

2 VISION TRANSFORMER ENCODER ON ACTIVE RECONFIGURABLE ARRAYS

2.1 Motivation - Core Problem Addressed

The vast majority of deep neural networks today utilize the seminal transformer architecture [5], which utilizes the concept of attention. Vision Transformers (ViT) are a powerful model for image recognition, and are extremely important in applications such as robot vision, autonomous vehicles, etc. which use real-time edge inference. However, their core multi-head self-attention (MHA) layers involve large matrix operations and are memory-intensive. The MHA is computationally intensive: as the number of heads and patches increases, the volume of matrix multiplications and nonlinear Softmax operations grows rapidly. High throughput and on-chip memory bandwidth is also demanded. Thus, running a ViT on edge hardware (e.g. FPGA-like devices) is challenging due to this high compute and memory demand. The ViT is the encoder of a transformer (followed by a classifier), which consists of a sequence of attention and feed-forward neural network blocks with residual connections across attention layers. The aim is to accelerate this network on an active reconfigurable array processor for edge applications.

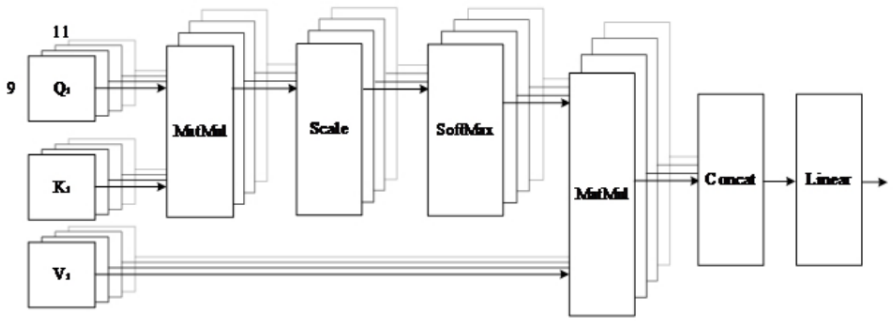

Fig. 4. Multi-Headed Attention Process [4]

2.2 Contributions

- To address low hardware utilization and high power, a specialized hardware encoder architecture is proposed, based on a dynamically reconfigurable processor array. The Processing elements have a pipeline-based utilization to make best use of the dataflow in the transformer (alternating attention and FFNN layers) with minimal wastage of hardware.
- In the proposed design, all four attention heads execute concurrently on separate Processing Element Groups, with fast data exchange and control via an instruction-driven reconfiguration mechanism.
- This scheme parallelizes the 4-head Q-K-V computations and introduces algorithmic improvements to the Softmax and data flow. Normally, the Q-K-V matrices are calculated separately for each head.

2.2.1 Changes to multi-head attention calculation. A major algorithmic contribution in this paper is optimizing the softmax and the data layout for attention, by introducing a hardware-friendly scheme that uses the log-sum-exp trick to avoid direct exponent/division units. By subtracting the maximum element and computing exponentials and sums in a reduced-precision or iterative manner, the Softmax unit avoids expensive multipliers and lookup tables. This reduces numerical error and critical path delay. In this design, the Softmax computation is pipelined and overlapped

$$Q = Q_1 \times W_{Q_1} + Q_2 \times W_{Q_2} + Q_3 \times W_{Q_3} + Q_4 \times W_{Q_4} \quad (1)$$

$$K = K_1 \times W_{K_1} + K_2 \times W_{K_2} + K_3 \times W_{K_3} + K_4 \times W_{K_4} \quad (2)$$

$$V = V_1 \times W_{V_1} + V_2 \times W_{V_2} + V_3 \times W_{V_3} + V_4 \times W_{V_4} \quad (3)$$

$$Attention\,(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (4)$$

Fig. 5. Linear Mapping [4]

$$softmax\,(S_i) = \frac{e^{s_i}}{\sum_{n=1}^{N} e^{s_n}} \quad (5)$$

$$Attention\left(Q_1K_1^T, V\right) = \frac{Q_1K_1^T}{\sum_{i=1}^{I} Q_iK_i^T} \times V \quad (6)$$

$$Attention\left(Q_2K_2^T, V\right) = \frac{Q_2K_2^T}{\sum_{i=1}^{I} Q_iK_i^T} \times V \quad (7)$$

$$Attention\left(Q_3K_3^T, V\right) = \frac{Q_3K_3^T}{\sum_{i=1}^{I} Q_iK_i^T} \times V \quad (8)$$

$$Attention\left(Q_4K_4^T, V\right) = \frac{Q_4K_4^T}{\sum_{i=1}^{I} Q_iK_i^T} \times V \quad (9)$$

Fig. 6. Attention Computation [4]

with other work: the array computes the (unnormalized) exponentials and partial sums while V-multiplication proceeds, so that the normalized attention weights are ready by the time they are required.

2.2.2 Hardware Architecture.

- The hardware architecture is shown in the below figure. The core innovation is an *Active Reconfigurable Array Processor* (ARAP) tailored to the ViT encoder.
- It also consists of a Host Interface (HI) and a Global Controller (GC) that manage data and configuration, plus a 4×4 array of Processing Elements (PEs) organized into four 2×2 Processing Element Groups (PEGs). Each PEG can be configured independently.
- Here, one of the clusters (PEG00) is allocated to the MHA computations (computing the attention weight matrix QK^T and weight multiplication), while PEG10 (another cluster) handles the feed-forward neural network (FFNN) and residual additions. Thus, design sustains a high hardware utilization by ensuring that whenever one PEG group is computing Softmax or accumulating sums, another group is already working on the next linear operation.

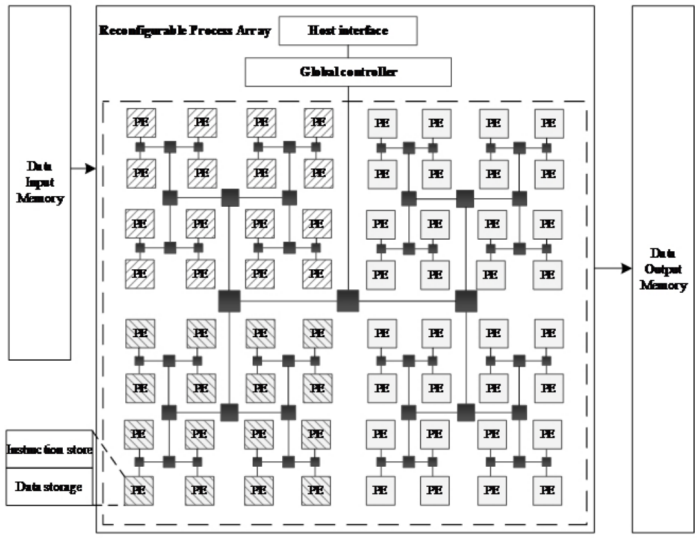

Fig. 7. Attention Computation [4]

- During operation, the GC issues a sequence instructions (44 bits long) that dynamically re-pipelines the array. For example, after computing the QK^T product, the array is reconfigured to perform the masked Softmax on each row, then reconfigured again to multiply by V, perform ReLU, etc.
- The design uses handshake signals to switch clusters: once the 4-way self-attention in the first cluster completes, it triggers reconfiguration of the second cluster, and vice versa. By time-multiplexing the hardware in this way, the overall design achieves parallel computation across all four heads with minimal stalls. We note that there are no timing conflicts and all dependencies are satisfied.

2.2.3 Dataflow.

- The input's channels are sequentially fed into the array, i.e. it performs attention on each channel in turns. This scheme ensure that at most one frame's data is in the array at a time, avoiding buffer overflows. It also aligns the data with the 4-way attention: each PEG computes one of the four heads on corresponding parts of the patch embeddings. This reorganization increases data locality and bandwidth.
- The dataflow graph shown below maximizes MHA parallelism based on data flow maps and makes best use of hardware. No direct division is performed, instead bit shift operations and adders are utilized for both numerators and denominators (giving slight approximation error).

2.3 Experimental Results

The proposed encoder was implemented on a Xilinx Versal XCVU440 (7nm) FPGA, and compared to prior ViT accelerators.

- **Resource reduction:** 17.09% fewer logic resources overall (e.g. DSPs/LUTs) compared to prior work.

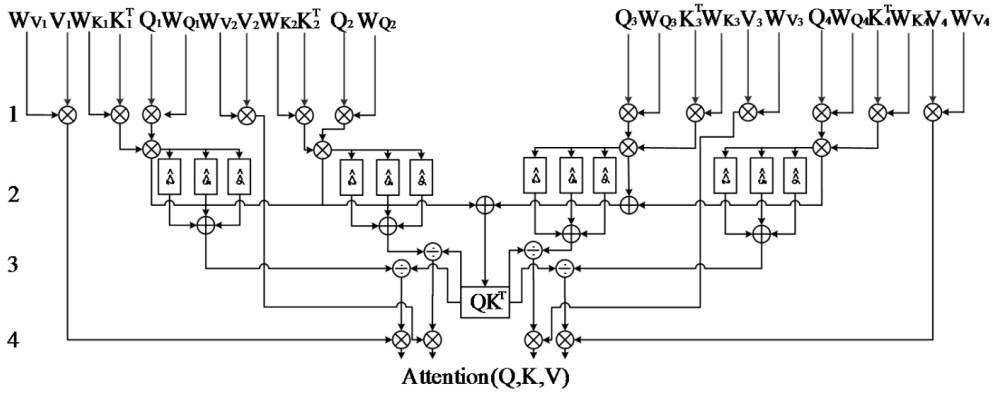

Fig. 8. Dataflow [4]

Hardware Resource	Ref.[10]	Ref.[11]	Ours
DSP	2,420	1,024 (512)	32
LUT	258,000	65,385 (132,433)	51,062
FF	257,000	31,739 (52,332)	32,757
BRAM	1,002	539 (439)	475
Platform	Alveo U50	ZCU102	XCVU440

Fig. 9. Results. Ref [10] includes Wang T et al. "Via" and Ref [11] includes Yang X et al. "EFA-Trans"

- **Accuracy improvement:** +0.61% higher classification accuracy on the ViT baseline
- **Latency savings:** 7.29% lower inference latency (5.85 ms) due to the parallel pipeline.

These gains occur because of the architectural innovations. By driving all 4 attention heads in parallel and carefully overlapping compute stages, the compute units remain busy. Shan and Zheng report that the array utilization in their MHA design approaches the theoretical maximum, substantially improving computational efficiency.

3 SWAPNET: EFFICIENT SWAPPING FOR DNN INFERENCE ON EDGE AI DEVICES BEYOND THE MEMORY BUDGET

3.1 Core Problem Addressed

Edge AI devices—such as NVIDIA Jetson modules in drones, robots, and autonomous vehicles—must run multiple, large-scale convolutional neural networks (CNNs) (e.g., YOLOv3, FCN, VGG-19, ResNet-101) alongside other sensor and system tasks, all within a very limited on-board memory (typically 2–8 GB). When the aggregate memory requirement of these models exceeds available DRAM/VRAM, inference either fails outright (out-of-memory errors) or forces designers to resort to model compression (pruning/quantization) that degrades accuracy, or to cloud offloading that adds unpredictable latency and compromises autonomy and privacy. SwapNet confronts this "**memory gap**" head-on by enabling full-accuracy, multi-CNN inference *beyond* the raw memory budget of the device—without retraining models or relying on network connectivity.

3.2 Architectural Innovations

SwapNet introduces a lightweight middleware layer between the deep-learning framework (e.g., PyTorch) and the GPU runtime, comprising two key controllers and a scheduler interface. . To enable large DNN inference beyond the memory budget, it partitions the DNN model into blocks, where each block consists of one or more neural network layers. Given a memory budget b allocated to a DNN model of size s (s>b), we aim to partition it into n blocks and execute them individually. Therefore, we need to decide the number of blocks n and the partition points, which we call the partitioning strategy. Once the partitioning strategy is obtained and the model is partitioned, the blocks are kept in the external storage, and are swapped in and out of the memory in order for DNN inference, which allows large DNNs to be executed within a small memory budget. The following are innovations convolved in the making of SwapNet .

3.2.1 ***Zero-Copy Block Swap-In Controller.***

- In standard swap-in, to load the DNN block data from the external storage to the memory of CPU and GPU, the read operation is called to load block data into the system memory for CPU access. If the computation is assigned to the GPU, a `dispatch` function is further called to copy the block data from the CPU memory area to the GPU memory area. This has two major drawbacks.
- When multiple models run concurrently in limited memory, the `read` operation copies blocks to the page cache, which can lead to high cache miss rates. Furthermore, due to the separation of CPU and GPU memory-since GPUs are often externally integrated-the memory addresses used by the CPU are not directly compatible with the GPU. As a result, the `dispatch` function must convert and copy the block into GPU memory, causing two copies of the same block to exist simultaneously in system memory. This process introduces significant additional latency and memory overhead.
- To overcome the inefficiency of standard swap-in, a novel *Zero-Copy Block Swap-In* scheme was designed. It consisted of a direct block fetch method to bypass the read operation and a copy-free GPU dispatch function to eliminate the CPU-to-GPU memory conversion and copy overhead.
- *Direct Block Fetch:* SwapNet uses Direct Block Fetch to bypass the inefficiencies of the page cache during read operations. By leveraging direct memory access (DMA) and direct I/O, it establishes a dedicated swap-in channel that transfers blocks from external storage directly into memory without intermediate copies. This approach ensures stable latency and allows the fetched blocks to be accessed by the CPU or dispatched to the GPU as needed.
- *Copy-Free GPU Dispatch:* SwapNet introduces a copy-free GPU dispatch technique to eliminate redundant memory copying and format conversion during inference on edge devices. Typically, frameworks like PyTorch allocate memory using `malloc` on the CPU, requiring data to be converted and copied to GPU memory via `.to('cuda')`. This overhead exists despite CPU and GPU sharing physical memory on edge devices due to separate logical address spaces. SwapNet resolves this by replacing `malloc` with `cudaMallocManaged`, enabling unified memory allocation accessible by both CPU and GPU. To implement this with minimal changes, SwapNet analyzes the framework's source code to build a function call graph related to CPU allocation (e.g., using keywords like cpu, alloc), identifies the lowest-level `malloc`, and substitutes it. This approach removes the need for explicit memory transfers, reduces latency, and generalizes across frameworks like TensorFlow and MNN.

3.2.2 ***Block Assembly by Reference.***

- The default model assembly workflow creates a dummy model with random weights as a placeholder, then loads actual parameters during execution. This approach doubles peak memory usage per block and incurs significant delays due to repeated instantiation and memory copying during frequent block swapping.
- *Model Skeleton by Reference:* Instead of maintaining a full dummy model, only the model's architectural skeleton (with pointers) is kept in memory, significantly reducing both latency and memory usage during model assembly. The skeleton object is lightweight and always in memory, while the actual model parameters are stored separately and only loaded when needed for execution.
- *Model Parameter Registration:* Parameters are stored as an array, and each pointer in the skeleton directly references its corresponding parameter by index, enabling fast and efficient model assembly without time-consuming element searches.

3.2.3 *SwapNet Utility: Multi-DNN Scheduling with Efficient Swapping.*

- *SwapNet Abstractions for Scheduling:* SwapNet represents each model block b_i by the tuple

$$b_i \equiv \left(s_i,\ \text{FLOPs}_i,\ d_i,\ t_i^{\text{in}},\ t_i^{\text{ex}},\ t_i^{\text{out}}\right),$$

  where
  - s_i = block size (bytes),
  - FLOPs_i = compute cost (floating-point operations),
  - d_i = depth index of block i within its network,
  - $t_i^{\text{in}}, t_i^{\text{ex}}, t_i^{\text{out}}$ = swap-in, execution, and swap-out delays.
- *Observing Linear Trends:* Empirical profiling (see Fig. 10) shows:

$$\begin{aligned} t_i^{\text{in}} &\propto s_i \quad \text{(I/O-bound)}, \\ t_i^{\text{ex}} &\propto \text{FLOPs}_i \quad \text{(compute-bound)}, \\ t_i^{\text{out}} &\propto s_i \quad \text{(I/O-bound)}, \end{aligned}$$

  with a mild dependence on depth d_i (later layers slightly slower to assemble).
- *Delay Estimation via Linear Regression:* To predict delays without per-block benchmarking, SwapNet fits simple linear models:

$$\begin{aligned} t_i^{\text{in}} &\approx \alpha_{\text{in}}\, s_i + \beta_{\text{in}}\, d_i + \gamma_{\text{in}}, \\ t_i^{\text{ex}} &\approx \alpha_{\text{ex}}\, \text{FLOPs}_i + \beta_{\text{ex}}\, d_i + \gamma_{\text{ex}}, \\ t_i^{\text{out}} &\approx \alpha_{\text{out}}\, s_i + \beta_{\text{out}}\, d_i + \gamma_{\text{out}}. \end{aligned}$$

  Here the coefficients $\{\alpha, \beta, \gamma\}$ are obtained by least-squares fitting on a small sample of profiled blocks. The high R^2 values (>0.95) confirm the validity of these linear relations.

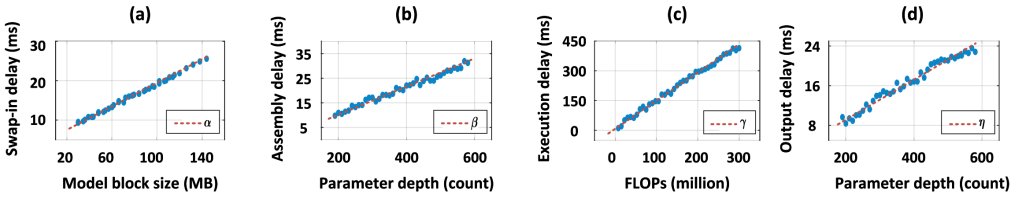

Fig. 10. Profiling the four device-dependent coefficients ($\alpha, \beta, \gamma, and \eta$) for the three delay components via linear regression.

- *Memory Constraint and Scheduling* With delay estimates in hand, a scheduler must interleave the events $\{\text{swap}-\text{in}(b_i), \text{exec}(b_i), \text{swap}-\text{out}(b_i)\}$ so that at all times (see Fig. 11)

$$\sum_{b_i \in \mathcal{L}(t)} s_i \;\le\; M$$

for memory budget M, and event start times follow

$$\text{start}(\text{exec}(b_i)) = \text{end}(\text{swap}-\text{in}(b_i)),$$
$$\text{start}(\text{swap}-\text{out}(b_i)) = \text{end}(\text{exec}(b_i)).$$

This abstraction lets any external scheduler optimize the makespan or per-model latency using the predicted delays.

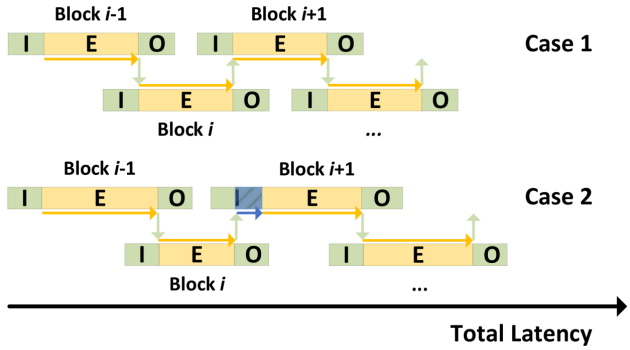

Fig. 11. Two cases of parallel execution of blocks for a given DNN.

- SwapNet's block-level API lets a simple scheduler interleave multiple DNNs under a shared memory cap M:

(1) **Per-Block Abstraction.** Each block b_i has size s_i, compute cost FLOPs_i, and estimated delays $t_i^{\text{in}}, t_i^{\text{ex}}, t_i^{\text{out}}$.

(2) **Global Memory Constraint.** At any time t, the sum of resident block sizes cannot exceed M:

$$\sum_{b_i \in \mathcal{L}(t)} s_i \;\le\; M.$$

(3) **Per-Model Quotas.** Divide M into per-model budgets $M^{(k)}$ (e.g. proportional to model importance), with $\sum_k M^{(k)} \le M$.

(4) **Block Partitioning.** For each model k, choose blocks so that $s_i \le M^{(k)}$, ensuring no single swap-in exceeds its quota.

(5) **Round-Robin Interleaving.** Cycle through models issuing, for each block b_i:

$$\text{swap-in}(b_i) \;\rightarrow\; \text{exec}(b_i) \;\rightarrow\; \text{swap-out}(b_i),$$

while checking $\sum s_j + s_i \le M$ before each swap-in.

(6) **Latency Prediction.** Use linear models

$$t_i^{(*)} \approx \alpha\, s_i + \beta\, \text{FLOPs}_i + \gamma$$

to estimate makespan or per-model latency and adjust quotas or block sizes as needed.

This scheme guarantees strict memory enforcement, predictable timing, and seamless integration with any external scheduler.

3.3 Evaluation Results on Edge Devices

SwapNet was implemented in PyTorch and tested on NVIDIA Jetson hardware (Jetson Xavier NX, 8 GB LPDDR4, 384-core Volta GPU) with models stored on NVMe SSD. Three scenarios were studied:

Table 1. SwapNet Evaluation: Memory Reduction and Latency Overhead

Scenario	Models & Sizes	Budget (MB)	Memory Reduction	Latency Overhead
Self-Driving	YOLOv3 (236 MB), FCN (207 MB), VGG-19 (548 MB), ResNet-101 (170 MB)	843	57–83 % per model	+14–47 ms (300–700 ms baseline)
Road-Side Unit (RSU)	2 × YOLOv3, 2 × ResNet-101, VGG-19 (1.4 GB total)	900	63–79 %	+22–55 ms
UAV Surveillance	YOLOv3 (236 MB), ResNet-101 (170 MB)	1000	64–75 %	+8–37 ms

- **Memory Compliance:** Models whose combined nominal size was up to 2.3×–5.8× the available memory ran without OOM.
- **Accuracy Preservation:** Inference accuracy matched the swap-free baseline.
- **Overhead:** SwapNet's metadata consumed only ~3–4 MB (~3.6% of 8 GB); power draw rose by $\approx 0.33W$.
- **Throughput:** A simple scheduler achieved end-to-end latency within 5–7% of an ideal plenty-of-memory baseline, despite concurrently running all models.

3.4 Key Contributions of SwapNet

- **Transparent Block-Swapping Middleware:** Introduces a layer-agnostic mechanism to partition arbitrarily large DNNs into blocks and swap them in and out of GPU memory at runtime, enabling on-device execution beyond the native memory limit.
- **Zero-Copy Swap-In Scheme:** Leverages CUDA unified-memory APIs to map host-resident weight pages directly into the GPU address space, eliminating explicit `cudaMemcpy` latency and reducing swap-in cost to near zero.
- **Multi-DNN Concurrency on Edge Devices:** Exposes a simple block-level API (`load_block`, `run_block`, `unload_block`) that enables interleaved inference across multiple oversized

models, allowing edge AI devices to run several DNNs simultaneously with near-native aggregate latency.

- **Empirical Validation on Edge AI Tasks:** Demonstrates the ability to handle 2.3–5.8× memory overcommit on real workloads with only a ~6.2% average latency overhead and up to 7.3% higher accuracy compared to compression-based baselines, all with minimal changes to existing PyTorch/CUDA code.

4 HERMES: DEEP DIVE

4.1 Core Problem Addressed

Modern transformer models (e.g. BERT-Large, ViT-Large, GPT-2, GPT-J) can easily consume gigabytes to tens of gigabytes of memory at inference time—far more than the 2–8 GB available on typical edge devices. Naively streaming model weights in and out of RAM stalls the CPU for I/O, and loading entire models into memory simply isn't possible. As a result, large-scale transformers remain confined to servers or the cloud, preventing low-latency, privacy-preserving inference on-device. Hermes tackles this *memory bottleneck* by orchestrating a fine-grained, pipelined execution that (a) never holds the whole model in RAM at once and (b) overlaps slow I/O with compute to hide load latency.

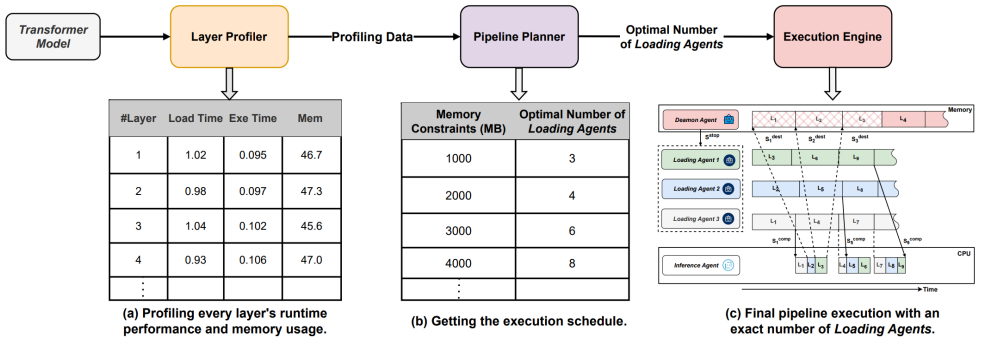

Fig. 12. Hermes system architecture: Layer Profiler, Pipeline Planner, and Execution Engine.

4.2 Architectural Innovations

Hermes is a unified runtime built around the **PipeLoad** execution mechanism, with three collaborating components:

- **Layer Profiler:**
  - Performs a one-time pass over the model to record per-layer weight size, disk-load time, and CPU compute time.
  - Outputs a table of per-layer statistics used by the planner.
- **Pipeline Planner:**
  - Given a memory budget, searches for the optimal number of *Loading Agents* to spawn.
  - Trades off *more agents* (finer granularity, lower latency, higher concurrent memory) vs. *fewer agents* (coarser granularity, higher latency, lower peak memory).
- **Execution Engine:**
  - *Loading Agents*: Each asynchronously loads a disjoint subset of layers into RAM.
  - *Inference Agent*: Consumes layers in order as soon as they are ready, performing the forward pass.

– *Daemon Agent*: Frees each layer's weights immediately after use, reclaiming memory for new loads.

These 3 agents work in unison to yield an effective pipeline.

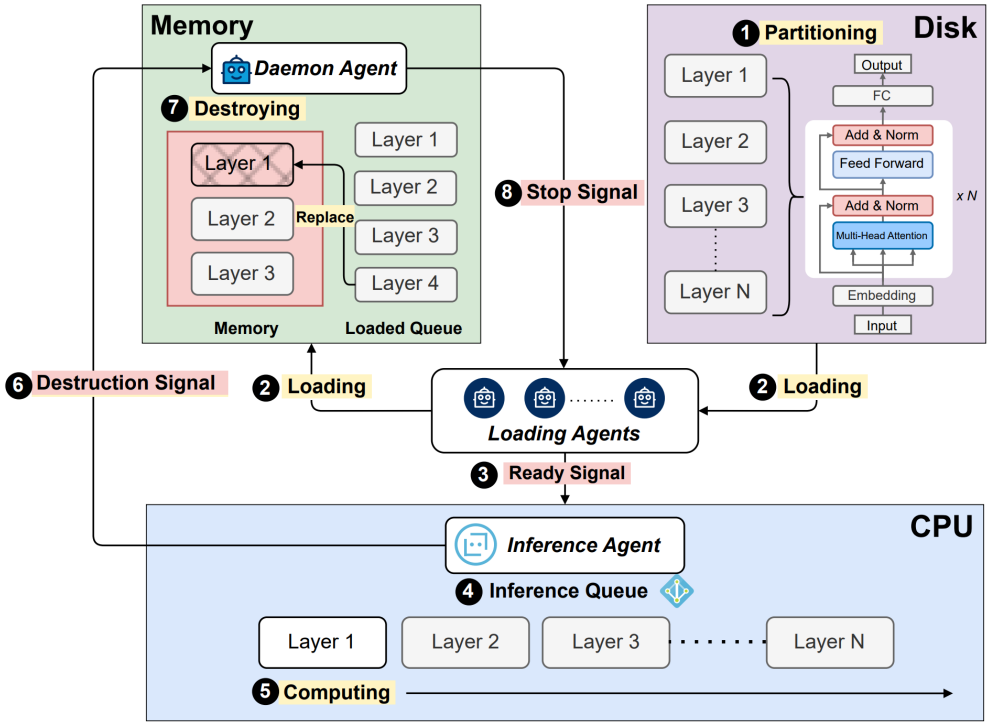

Fig. 13. PipeLoad workflow with Loading Agents, Inference Agent, and Daemon Agent coordinating load–compute–free.

(1) **Layer Partitioning (Stage 1, Fig. 13):** Pre-process the model by partitioning its weights at layer granularity.
(2) **Parallel Loading (Stage 2):** Multiple *Loading Agents* asynchronously fetch their assigned layers from disk into RAM.
(3) **Ready Signaling (Stage 3):** Once a layer is loaded, the corresponding Loading Agent sends a "compute-ready" signal for that layer to the Inference Agent.
(4) **Inference Queue Management (Stage 4):** The *Inference Agent* maintains a queue of ready layers, preserving the original layer sequence.
(5) **Layer Computation (Stage 5):** Upon receiving the ready signal—and after all preceding layers have been computed—the Inference Agent performs the forward pass on that layer.
(6) **Destruction Signaling (Stage 6):** After computing a layer, the Inference Agent issues a memory-destruction signal to the Daemon Agent.
(7) **Memory Reclamation (Stage 7):** The *Daemon Agent* frees the RAM occupied by the completed layer to make room for new loads.
(8) **Flow Control (Stage 8):** If total memory usage approaches or exceeds the device's limit, the Daemon Agent broadcasts a stop signal to all Loading Agents, pausing further loads until enough memory is reclaimed.

This design yields a *fine-grained pipeline* in which I/O and compute are aggressively overlapped, and the working set never exceeds the chosen memory cap.

4.3 Evaluation Results on Constrained Hardware

Hermes was evaluated on four representative transformer models under an 8-core, CPU-only environment (Docker-capped RAM) to emulate edge devices. Results for the *6 Loading Agent* configuration are summarized in Table 2.

Table 2. Hermes Evaluation: Memory Footprint and Latency Improvements

Model	Baseline RAM	Hermes RAM	Reduction	Latency (Base→Hermes)
BERT-Large (340M)	1627 MB	931 MB	43 %	15.9 s→3.51 s (4.5×)
ViT-Large (304M)	600 MB	159 MB	74 %	345 ms→43 ms (8.0×)
GPT-2 Base (355M)	1400 MB	563 MB	60 %	1.66 s→1.12 s (1.48×)
GPT-J (6B)	12.35 GB	3.24 GB	74 %	31.33 s→29.64 s (1.06×)

Key Observations:

- **Memory Compliance:** Even the 12 GB GPT-J model fits under a 4 GB cap when using 6 agents.
- **Latency Gains:** Encoder-only models (BERT, ViT) see 4–8× speedups; autoregressive GPT models see modest gains.
- **Adaptivity:** Under tighter RAM budgets, the Planner reduces agent count to trade latency for lower peak memory.

4.4 Key Contributions of Hermes

- **Transformer Support on Edge:** Enables the deployment of transformer-based models (e.g., BERT, Vision Transformer) on memory-constrained edge AI devices by dynamically swapping attention and feed-forward blocks, without sacrificing real-time inference capabilities.
- **Layer-Level Profiler:** Implements fine-grained monitoring of per-layer memory usage and execution latency, enabling the system to characterize "hot" and "cold" tensors and to anticipate memory demands before kernel dispatch.
- **Pipeline Planner:** Constructs an optimized execution and swap schedule by analyzing profiling data, arranging layers into a memory–compute pipeline that minimizes fragmentation impacts and balances swap frequency against computation flow.
- **Adaptive Execution Engine:** Orchestrates on-demand swap-in/out and compute tasks, dynamically adjusting swap thresholds and block granularity in response to runtime deviations, thereby reducing swap-induced stalls by up to 50%.
- **Fragmentation-Aware Memory Management:** Integrates with underlying allocators to provision contiguous memory regions for large tensors, mitigating allocation failures due to fragmentation and ensuring predictable low-latency allocation.
- **Seamless Framework Integration:** Provides minimal-intrusion hooks into TensorFlow and PyTorch memory allocators, requiring no changes to user model definitions while delivering fragmented-aware, adaptive DNN inference on edge AI devices.

5 CONCLUSION AND FUTURE DIRECTIONS

Conclusions. All four works—Zhang *et al.*'s DVFO, Shan and Zheng's ViT accelerator, Wang *et al.*'s SwapNet, and Xueyuan Han's Hermes—demonstrate that pushing ever-larger and more

complex DNNs onto resource-constrained edge devices requires a holistic, co-optimized stack spanning hardware, system software, and the neural network itself. Zhang *et al.* show that a DRL-driven DVFS and offloading controller can dynamically steer energy and latency trade-offs without manual tuning. Shan and Zheng prove that custom, reconfigurable architectures can exploit the innate parallelism of transformer blocks to achieve high throughput under tight resource budgets. SwapNet introduces a transparent block-swapping middleware—complete with zero-copy swap-in and pointer-based graph assembly—enabling arbitrarily large CNNs (or any DNN) to run on unified-memory devices and even interleave multiple models concurrently. Hermes complements these approaches by adding fine-grained layer profiling, pipeline planning, and an adaptive execution engine that collectively tame memory fragmentation and runtime jitter, extending low-latency, predictable inference support to transformer-based models on the edge.

Future Directions. Building on these complementary advances, several promising avenues emerge. First, *cross-layer co-optimization* could unify DVFS/offload controllers with SwapNet's and Hermes's profiling data, allowing an RL agent to jointly adjust clock frequencies, swap thresholds, block sizes, and offloading decisions in one end-to-end loop. Second, *dynamic model adaptation*—for example, skipping or compressing layers when energy budgets tighten—could leverage real-time telemetry to reshape the DNN itself in flight. Third, *generalized reconfigurable accelerators* that switch between CNN, RNN, and Transformer modes would extend Shan & Zheng's hardware specialization toward multiparadigm fabrics, with scheduling informed by Hermes's pipeline planner. Fourth, *predictive resource scheduling*—using vision- or sensor-based workload forecasts—could proactively rescale memory regions, clock domains, and network links to smooth QoS under bursty conditions. Finally, *automated hardware-software co-design tools* could take a high-level model and device profile, then synthesize the optimal combination of DVFS policies, swap layouts, pipeline stages, and accelerator configurations, laying the groundwork for truly autonomous, self-optimizing edge-AI systems.

REFERENCES

[1] J. Dass et al. Vitality: Unifying low-rank and sparse approximation for vision transformer acceleration with a linear taylor attention. In *Proc. 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[2] Xueyuan Han, Zinuo Cai, Yichu Zhang, Chongxin Fan, Junhan Liu, Ruhui Ma, and Rajkumar Buyya. Hermes: Memory-efficient pipeline inference for large models on edge devices. In *Proceedings of the 42nd IEEE International Conference on Computer Design (ICCD)*, pages 454–461, 2024.

[3] Naveen R and Sudhish N George. Power efficient asic design for vision transformer using systolic array matrix multiplier. In *28th International Symposium on VLSI Design and Test*, 2024.

[4] Rui Shan and Yifan Zheng. Design and implementation of encoder for vision transformer networks based on active reconfigurable array processors. In *Proceedings of the 2023 6th International Conference on Artificial Intelligence and Pattern Recognition (AIPR)*, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, volume 30. Curran Associates, Inc., 2017.

[6] Kun Wang, Jiani Cao, Zimu Zhou, and Zhenjiang Li. Swapnet: Efficient swapping for dnn inference on edge ai devices beyond the memory budget. *IEEE Transactions on Mobile Computing*, 23(9):8935–8950, 2024.